Fault Isolation using Process Algebra Models

Dan Lawesson[1], Ulf Nilsson[1], Inger Klein[2]
[1]Dept of Computer and Information Science
[2]Dept of Electrical Engineering
Linköping University, 581 83 Linköping, SWEDEN
{danla,ulfni}@ida.liu.se inger@isy.liu.se

Abstract

We investigate the problem of doing post mortem fault isolation for concurrent systems using a behavioral model. The aim is to isolate the action that has caused the failure of the system, the root action. The naive approach would be to say that a certain action is the root action iff it is a logical consequence of the model and observations that the action is the first "bad thing to happen". This, however, is a strong requirement and puts high demand on the model. In this paper we describe the concept of *strong root candidate*, a relaxation of the naive approach. The advantage of determining the strong root candidate directly from model and observations is that the set of traces consistent with model and observations need not be explicitly computed. The property of strong root candidate can instead be determined on-the-fly, thus only computing relevant parts of the reachable state space.

1 Introduction

In this paper we describe a model-based [Hamscher *et al.*, 1992] approach to fault isolation in object oriented control software. The work is motivated by a real industrial robot control system developed by ABB Robotics. The system is large (the order of 10^6 lines of code), concurrent, has an object oriented architecture and is highly configurable, supporting different types of robots and cell configurations. Since the system is time- and safety-critical the first priority, in case of a failure, is to bring the system to a safe state; alarms that go off are logged and can be analyzed when the system comes to a stand-still. The faults considered are primarily hardware faults, and therefore we rely on the assumption that the failing hardware has some software counterpart that is affected by the failure of the hardware. In addition we make the common single fault assumption, i.e. that a system failure is caused by only one fault (but resulting in cascading alarms).

The log thus contains *partial information* about the events that took place at the approximate time of the system failure. However, the order in which messages are logged does not necessarily reflect the way error messages propagate – the system is concurrent and safety critical actions may have to be taken before error reporting takes place. Hence, in what follows we (somewhat conservatively) view the log as a *set* of error messages. In addition a system may contain a number of critical events that are unobservable, but which may explain all observable alarms.

The ultimate aim of our fault isolation method is to single out the error message that explains the actual cause of the failure, or possibly an unobservable critical event explaining the observations. That is, we aim to discard error messages which are definitely effects of other error messages, while trying to isolate error messages (or critical events) which explain all other messages. In contrast to message filtering, we can thus find failing components that have not manifested themselves in the error log, if the failing of the component is a logical consequence of the model and the observations. Given the size of the software it is not possible to use the code directly – we have to rely on a model of the software. In this paper we consider finite state machine models expressed in a process algebra. The process algebra is chosen here because it allows for more straightforward formal reasoning than for example state charts, but the contribution of this work - the fault isolation - relies only on the labeled transition system semantics of the model. In practice, the aim is to use a behavioral model that is an artifact of the software development process, such as state charts. Then there is no extra cost associated with maintaining a correct model when the software evolves, since then so does the model.

In standard AI diagnosis literature, see e.g. [Reiter, 1987], a diagnosis is a (minimal) set of failed components explaining the observations. But for dynamic systems (systems with state) a diagnosis is often defined as the set of all traces, or trajectories, consistent with the observations (see e.g. [Cordier *et al.*, 2001; Console *et al.*, 2000]). However, this definition is generally insufficient to isolate the origin of the fault(s), and requires post-processing to pin-point e.g. the faulty component(s). Our approach is more direct and focuses on finding the alarm that explains (is consistent with) all observables: given the system description, expressed in a simple process algebra, and the observations, we try to infer the origin of the fault using properties of actions involving the temporal order, expressed in a specification language based on a subset of the temporal logic CTL, originally developed for verification [Clarke *et al.*, 1999]. This resembles the process of model checking and as in the case of model-checking there is no need for calculation of the entire state space (obviously

equivalent to the set of traces consistent with model and observations) if the temporal logic formulae are evaluated by constructing the state space on-the-fly.

Our approach also bears some resemblance to that of Sampath et al. [Sampath *et al.*, 1995]. However their work is mainly concerned with diagnosability in discrete event systems; i.e. to detect, within finite delay, whether a certain type of fault has occurred. While our approach is amenable only to post-mortem analysis, the work reported in [Sampath *et al.*, 1995] is mainly intended for monitoring and on-line detection and diagnosis.

The rest of the paper is organized as follows: In Section 2 we describe the behavior language that will be used to define a transition relation, that defines the set of all possible behaviors (i.e. traces). In Section 3 we provide rules for entailment of some predicates of interest from configurations and the traces that can follow from them. Finally, we outline ongoing and future work in Section 4.

2 A behavior language

A behavior model can be expressed in different ways, and we have chosen to use a process algebra. No matter which formalism and notation that is used, the semantics should provide a labeled transition relation that describes the state transitions of the modeled system. In this section we describe a process algebra influenced by CCS [Milner, 1989] and give the necessary semantics.

2.1 Processes

Our process language is constructed from the following syntactic categories

- a finite set $\mathcal{L}$ of *action labels* denoted by a in our meta language. Every action label is equipped with an associated arity $n \geq 0$.
- a set $\mathcal{O}$ of *object id's* denoted by o.
- a finite set $\mathcal{S}$ of *states* A with associated arity $n \geq 0$.

We consider four types of *actions* (denoted by α in our meta language).

- Send actions of the form $o{:}\overline{a}(\mathbf{t})$, where o is the recipient object, $\overline{a}$ an n-ary action label and $\mathbf{t}$ is an n-tuple of object id's or variables.
- Receive actions of the form $a(\mathbf{x})$ where a is an n-ary action label and $\mathbf{x}$ is an n-tuple of variables.
- Internal actions of the form a, where a is a nullary action label.
- New-actions of the form $new(o, P)$ where $o \in \mathcal{O}$ and P is a process expression, defined below.

A process is described by a *process expression*, denoted by P (and occasionally Q), and given by the following abstract syntax

$$\mathcal{P} ::= A(\mathbf{t}) \mid \sum_{i \in I} \alpha_i.\mathcal{P}$$

where I is a finite index set. Sums are usually written simply $\alpha_1.P_1 + \alpha_2.P_2$. We reserve the nullary state *Stop* for a completed process. We assume that every $A/n \in \mathcal{S}$ (*Stop* excepted) has a defining equation of the form

$$A(\mathbf{x}) \stackrel{def}{=} P.$$

A *process state* σ is a partial map from $\mathcal{O}$ to $\mathcal{P}$. The object $init \in \mathcal{O}$ is called the *initializing object*, the state $Main \in \mathcal{S}$ is called the *main process* and the state $\sigma_0 := \{init \mapsto Main\}$ is called the *initial process state*.

Let $\sigma{:}\mathcal{O} \to \mathcal{P}$ be a process state, $o \in \mathcal{O}$ and $P \in \mathcal{P}$. By $\sigma[o \mapsto P]$ we denote the process state which is almost identical to σ except possibly at o. That is

$$\sigma[o \mapsto P](x) := \begin{cases} P \text{ if } x = o \\ \sigma(x) \text{ otherwise} \end{cases}$$

The behaviors of our system are described by the labeled transition rules in Figure 1. Our transitions are of the form

$$\sigma \xrightarrow{\alpha} \sigma'$$

where α (the observation) is a set of pairs of the form (o, a) representing action a occurring in object o.

There are four transition rules, *sync*, *internal*, *new* and *def*. The rule *sync* allows two objects to synchronize their state transitions and optionally exchange values. In our limited algebra, the only values that can be transmitted are object identifiers. However, the idea is not to model all system behavior, but to have a system model that reveals synchronization and system structure. The rule *internal* allows a single object to perform a transition by itself. Creation of new objects is handled by the rule *new*, and *def* allows for exchanging a state with its definition.[1]

Example

Typically, a system is described by creating a main process that sets up the system structure. Figure 2 shows an example of such a system. Process $Main$ creates three objects and runs $Setup$ which tells the objects about each other via the $\overline{init}$ call. This is needed since when started, a process does not know anything about its environment. After $init$, each object will act as a peer-to-peer node, as showed in Figures 3 (the system) and 4 (object details). Objects can send requests to each other, and sometimes the answer to a request is a failure, and then the system is brought to a halt by transmission of $down$ messages.

3 Fault Isolation

The available information when doing fault isolation is a system model and an observation (in our case a message log). We use the term *scenario* to refer to that information. In the following we overload the term action in the context of scenarios to mean pairs $(o, a) \in \mathcal{O} \times \mathcal{L}$ where o is an object identifier and a is an action label. Some of the actions in a system are *critical actions*, actions that are associated with system failures.

Thus a scenario is a quadruple $(\longrightarrow, Crit, Log_p, Log_n)$, where $\longrightarrow$ is a process state transition relation, $Crit \subseteq \mathcal{O} \times \mathcal{L}$

[1] Since we rely on a finite state space model, we do not allow unbounded creation of objects via the *new* rule.

$$\text{sync}: \frac{\sigma(o_i) = P_1 + o_j : \overline{a}(\mathbf{t}).P + P_2 \quad \sigma(o_j) = P_3 + a(\mathbf{x}).Q + P_4}{\sigma \xrightarrow{\{(o_i,\overline{a}),(o_j,a)\}} \sigma[o_i \mapsto P][o_j \mapsto Q\{\mathbf{x}/\mathbf{t}\}]}$$

$$\text{internal}: \frac{\sigma(o_i) = P_1 + a.P + P_2}{\sigma \xrightarrow{\{(o_i,a)\}} \sigma[o_i \mapsto P]}$$

$$\text{new}: \frac{\sigma(o_i) = P_1 + new(o, Q).P + P_2 \quad \sigma[o_i \mapsto P][o \mapsto Q] \xrightarrow{\alpha} \sigma'}{\sigma \xrightarrow{\alpha} \sigma'}$$

$$\text{def}: \frac{\sigma(o_i) = A(\mathbf{t}) \quad A(\mathbf{x}) \stackrel{def}{=} P \quad \sigma[o_i \mapsto P\{\mathbf{x}/\mathbf{t}\}] \xrightarrow{\alpha} \sigma'}{\sigma \xrightarrow{\alpha} \sigma'}$$

Figure 1: Process transition rules ($\mathbf{t}$ is a vector of object id's)

$$\begin{array}{lll}
Servent(this, x, y) & \stackrel{def}{=} & x{:}\overline{req}(this).Wait(this, x, y) + y{:}\overline{req}(this).Wait(this, x, y) + \\
 & & req(o).Compute(this, x, y, o) + down().Down \\
Wait(this, x, y) & \stackrel{def}{=} & ok().Servent(this, x, y) + fail().Fail(x, y) \\
Compute(this, x, y, o) & \stackrel{def}{=} & o{:}\overline{ok}().Servent(this, x, y) + o{:}\overline{fail}().Servent(this, x, y) \\
Fail(x, y) & \stackrel{def}{=} & x{:}\overline{down}().Fail(x, y) + y{:}\overline{down}().Fail(x, y) \\
Down & \stackrel{def}{=} & Stop \\
S & \stackrel{def}{=} & init(this, x, y).Servent(this, x, y) \\
Main & \stackrel{def}{=} & new(s_1, S).new(s_2, S).new(s_3, S).Setup(s_1, s_2, s_3) \\
Setup(x, y, z) & \stackrel{def}{=} & x{:}\overline{init}(x, y, z).y{:}\overline{init}(y, z, x).z{:}\overline{init}(z, x, y).Stop
\end{array}$$

Figure 2: A process algebra example

is the set of critical actions, $Log_p \subseteq \mathcal{O} \times \mathcal{L}$ is the set of actions that have been observed (i.e. the message log), and $Log_n \subseteq \mathcal{O} \times \mathcal{L}$ is the set of actions known not to have occurred (i.e. the observable actions not contained in the message log). Thus, we assume that a synchronized action is logged as two separate actions – one from the sending object and one from the receiving. This allows modeling of message sending with unknown receiver and is no severe limitation since it is possible to express receiver information by having a model where the desired action labels are unique and receiver object id thus becomes unambiguous.

A *configuration*, denoted C, is the symbol $\perp$ or a pair (σ, l) where σ is a process state and $l \subseteq \mathcal{O} \times \mathcal{L}$ is a set of actions. The following rules defines the *configuration transition relation* $\Rightarrow$ for a given $\longrightarrow$ and Log_n.

$$\frac{\sigma \xrightarrow{\alpha} \sigma' \quad \alpha \cap Log_n = \emptyset}{(\sigma, l) \Rightarrow (\sigma', l \cup \alpha)}$$

$$\frac{\sigma \xrightarrow{\alpha} \sigma' \quad \alpha \cap Log_n \neq \emptyset}{(\sigma, l) \Rightarrow \perp}$$

The configuration $(\{init \mapsto Main\}, \emptyset)$ is called the *initial configuration*. The configuration $\perp$ is called *a forbidden configuration* and represent configurations that are allowed by the behavioral model, but inconsistent with the observations at hand. We see configurations as snapshots of the system state of a given scenario, and the configuration transition relation describes the behavior of the system. Fault isolation is the process of finding the first critical action that has occurred in a given scenario, the *root action*. Given the single fault assumption and a system model that is properly designed, the first critical action to occur in the system is the cause of the failure.

An action α is *present* in a scenario if the system model and the observation entails the occurrence of α. An action α is an *enabled root* if the assumption that α is root action is consistent with the observations and the system model. We introduce the concept of *strong root candidate*, and say that a strong root candidate is an action that is both present and an enabled root.

3.1 Predicate rules

Given a certain scenario $(\rightarrow, Crit, Log_p, Log_n)$, we wish to reason about properties of reachable configurations. Therefore we define predicates, that correspond to the interesting properties, by determining for which configurations they hold true. Since we are interested in strong root candidates, we need to formally define present actions and enabled root actions. Thus we define the predicate $present(\alpha)$ that holds in configurations where action α must occur sometime in the future and the predicate $enabledroot(\alpha)$ that holds for configurations where it is consistent to assume that α may be the first critical action to occur. In defining these two predicates, we will need some helper predicates. We will use $okend$ that holds in configurations that correspond to consistent ending states of the system. An ending state is a state where no more observable actions occur, i.e. when the system has reached a final state. In a configuration where α has occurred, $seen(\alpha)$ holds, while $nocrit$ holds in configurations where no critical action has occurred. The predicate end holds in configurations where there is no next configuration.

We define entailment of logical formulae from the following syntax:

$$\mathcal{F} ::= \mathcal{F} \vee \mathcal{F} \mid \mathcal{F} \wedge \mathcal{F} \mid \neg\mathcal{F} \mid EF(\mathcal{F}) \mid EX(\mathcal{F}) \mid AG(\mathcal{F}) \mid end \mid okend \mid nocrit \mid seen(\alpha) \mid present(\alpha) \mid enabledroot(\alpha)$$

In order to be able to define entailment for the desired predicates, we will need the following. We use $\stackrel{*}{\Rightarrow}$ for the reflexive transitive closure of $\Rightarrow$. First we define entailment for basic connectives.

$$\frac{C \models F_1 \quad C \models F_2}{C \models F_1 \wedge F_2} \qquad \frac{C \models F_2}{C \models F_1 \vee F_2}$$

$$\frac{C \models F_1}{C \models F_1 \vee F_2} \qquad \frac{C \not\models F}{C \models \neg F}$$

We will be reasoning about temporal order, so we need to define temporal logic operators.

$$\frac{C \stackrel{*}{\Rightarrow} C' \quad C' \models F}{C \models EF(F)}$$

$$\frac{C \Rightarrow C' \quad C' \models F}{C \models EX(F)}$$

$$\frac{C' \models F \text{ whenever } C \stackrel{*}{\Rightarrow} C'}{C \models AG(F)}$$

We also need entailment for a few helper predicates. The predicate end determines if a configuration lacks successor (i.e. $end \equiv \neg EX(true)$ where $true$ is entailed by every configuration), $seen(\alpha)$ is true when an action α has occurred and $nocrit$ holds when no critical actions have yet occurred.

$$\frac{\neg\exists C', C \Rightarrow C'}{C \models end} \qquad \frac{\alpha \in l}{(\sigma, l) \models seen(\alpha)} \qquad \frac{\forall \alpha \in l, \alpha \notin Crit}{(\sigma, l) \models nocrit}$$

Now we have the tools needed to define the desired predicates. If we have reached a configuration from which the system cannot continue to execute and all actions in Log_p are seen, then the configuration is an $okend$, unless the configuration is a forbidden configuration. It is thus one of the possible halting configurations, given the scenario at hand.

$$\frac{\forall \alpha \in Log_p, C \models seen(\alpha) \quad C \models end \quad C \neq \perp}{C \models okend}$$

If it is true for all reachable configurations that whenever we have reached an $okend$, we have seen action α, we conclude that the presence of α is entailed from observations and system model.

$$\frac{C \models AG(\neg okend \vee seen(\alpha))}{C \models present(\alpha)}$$

If there is a reachable configuration C_1 such that no critical actions has taken place, and there is a configuration step that takes us from C_1 to C_2 where the critical action α has occurred, we conclude that α is an enabled root if it is possible to reach an $okend$ from C_2.

$$\frac{\alpha \in Crit \quad C \models EF(nocrit \wedge EX(seen(\alpha) \wedge EF(okend)))}{C \models enabledroot(\alpha)}$$

3.2 Reasoning about behavior

Given a scenario, the strong root candidates are the actions α for which

$$(\{init \mapsto Main\}, \emptyset) \models present(\alpha) \wedge enabledroot(\alpha)$$

If we have no strong root candidates or more than one strong root candidate, the system model is not strong enough for efficient fault isolation. If, on the other hand, we have exactly one strong root candidate, we assume that we have pinpointed the true cause of the fault. This is reasonable to assume, since the action found is the only one that is known to have occurred (its presence is entailed by the scenario) and it is consistent with the given scenario to assume that the action is a root event.

Of course there is still a possibility that there are other enabled root events whose presence are consistent with the scenario, but assuming one of them to be root would demand an explanation to why the strong root candidate (proven to be present!) is not the root.

3.3 Prototype implementation

We have designed a prototype XSB [Sagonas *et al.*, 1994] program that takes a system model and observations as input and enumerates the strong root candidates. XSB is a Prolog dialects using tabulation (memoization) to improve termination. Given the system model in Figure 2 and facts stating that any sending of *fail* or *down* indicates system failure, i.e. those actions are critical actions, and the observations that $(o_2, \overline{fail})$ *has not* occurred and $(o_3, fail)$ *has* occurred, the XSB Prolog program computed $(o_1, \overline{fail})$ to be the single strong root candidate.

The system consists of three objects that all execute the same process. See Figure 4 for an automata representation of a similar process (parameters are not explicit in the automata). Consider the critical actions. Obviously, no $\overline{down}$ message can be root action since it will always be preceded by a $\overline{fail}$ action, and neither can $(o_2, \overline{fail})$ be root action since it is known to not have occurred at all. This leaves us with $(o_1, \overline{fail})$ and $(o_3, \overline{fail})$. It is consistent with the system model and the observations to assume that $(o_3, \overline{fail})$ is the root action, since if o_2 receives the *fail* from o_3, then o_1 can send *fail* to o_3 afterwards. We cannot prove that $(o_3, \overline{fail})$ has happened, however. This can be done for $(o_1, \overline{fail})$, and therefore it is the only action that is both enabled root and present.

Thus, having some intuition of the system makes the fault isolation described above almost trivial, but the key motivation of this work is to formalize and automate this intuition.

4 Future Work

In previous work with Larsson [Larsson *et al.*, 2000; Larsson, 1999] we studied the fault isolation problem using a structural model. A key feature of that approach is the use of software engineering models, in particular UML [Object Management Group, 1999] class diagrams. Such a model can be developed and maintained at a relatively low cost being an integrated part of the software development process. The work presented here and in our previous work [Lawesson, 2000; Lawesson *et al.*, 2001] aims to strengthen the diagnostic capability while still using standard and state-of-the-art modeling notations. Behavior in UML is often expressed using statecharts, and process algebras provide a textual representation of state machines. Of course, enforcing the software developer to construct complete statecharts for all classes is not realistic in large software systems; hence, reasoning must be able to cope with incomplete or missing behavioral descriptions. Our approach should also be extended to deal with the special features characteristic of object oriented software systems such as classes and inheritance. Below we sketch some partial solutions to such issues, which will be addressed in our future work.

4.1 Classes behaviors and inheritance

Our process algebra expresses a system model as a flat set of the process defining equations without any hierarchy. In an object oriented design, the system behavior is partitioned into classes. Furthermore, inheritance allows for a hierarchy of classes. We implement simple schemas called classes in order to achieve the partitioning and (inheritance) hierarchy.

Thus, in the following a *class* is a scheme that can be compiled to a set of process defining equations. A class C may inherit parts of its characteristics (e.g. its behavior) from a *superclass*, and in that context C is referred to as the *subclass*. A state inheritance sequence

$$S \mapsto [A_1, A_2, ..., A_n]$$

is a declaration saying that state S in the superclass is refined by states $A_1, A_2, ..., A_n$ in the subclass where A_1 is the default state (i.e. the substate entered when entering the superstate S). When compiling the class to process equations, the inheritance sequence describes how the defining equations from the superclass should be used. Thus, we implement a simple form of inheritance as refinement. The syntax used for defining classes below is

$$N = (S, I), D$$

where N is the name of the class, S is the name of the superclass (if any), I is the set of state inheritance sequences and D is a set of process defining equations. If there is no superclass we write $N = (), D$.

Example

Lacking formal tools, we outline the approach by an example. In the following we define two classes C_1 and C_2, where C_2 refines the state A in C_1 with states C and D. We say that states C and D refine state A.

$$\begin{aligned}
C_1 = (), \{ & \\
& A \stackrel{def}{=} b.B \\
& B \stackrel{def}{=} a.A \\
\} & \\
C_2 = (C_1, \{A \mapsto [C, D]\}), \{ & \\
& C \stackrel{def}{=} d.D \\
& D \stackrel{def}{=} c.C \\
& B \stackrel{def}{=} e.D \\
\} &
\end{aligned}$$

Now, C_2 may be compiled to the following process equations.

$$C_2{:}C \stackrel{def}{=} b.C_2{:}B + d.C_2{:}D$$
$$C_2{:}B \stackrel{def}{=} a.C_2{:}C + e.C_2{:}D$$
$$C_2{:}D \stackrel{def}{=} b.C_2{:}B + c.C_2{:}C$$

The outgoing transitions from A become outgoing transitions from all refining states, while the incoming transitions are moved from the refined state to the first of the refining states. If there are transitions from the same state in both super- and subclass, they are joined as indeterministic choice, as with state B and transitions $a.A$ and $e.D$. The states are prefixed with the class name to avoid name space clashes.

4.2 Statecharts

Since both processes and statecharts have a transition system semantics, the mapping is straightforward once the semantics of the statecharts is fixed. We use a handshaking semantics of the statecharts, because of expressivity and domain properties as described in [Lawesson, 2000]. We define the semantics via our process language by providing a mapping from statecharts to processes. The mapping is rather straightforward since we restrict ourselves to statecharts without history states – essentially making the state chart equivalent to an automata without hierarchy, see for example [Lilius and Porres, 1999]. The process algebra example in Figure 2 could represent a slightly improved version[2] of the automata in Figure 4 with structure information (i.e. the states S, $Main$ and $Setup$) added.

4.3 Default behaviors of class diagrams

Since a class diagram in general does not contain behavioral information in terms of statecharts, we may introduce a superclass called *Propagator* that encapsulates the behavior of being able to propagate errors as well as reporting errors to the log, and a subclass *Breakable* that is a propagator that can introduce errors by the transition *crit*. The idea is to let all classes inherit from *Propagator*, and then refine with behavioral models when available, and use *Breakable* for classes that may give rise to critical actions but where a behavioral model is missing. The definition of *Propagator* and *Breakable* are given in Figure 5.

The paths of error propagation between classes is computed by using information about dependencies between classes in the class diagrams (as in [Larsson, 1999; Larsson *et al.*, 2000]), and then reflected in the $Failed(\mathbf{x})$ state that models error propagation.

Acknowledgments

This work has been financially supported by VINNOVA's Center of Excellence ISIS – Information Systems for Industrial Control and Supervision. We are also grateful for the cooperation with ABB Robotics, and in particular Magnus Larsson.

[2] In the process algebra version of this example a $\overline{req}$ also provides a reference to the caller, so that the called object know where to address the answer. This makes the example more similar to a real system, but in this completely symmetric case when the objects run the same code without parameters other than references to each others, the semantics remains the same.

References

[Clarke *et al.*, 1999] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[Console *et al.*, 2000] L. Console, C. Picardi, and M. Ribaudo. Diagnosis and Diagnosability Analysis using PEPA. In *Proc. 14th European Conference on Artificial Intelligence*, pages 131–136. IOS Press, 2000.

[Cordier *et al.*, 2001] Marie-Odile Cordier, Laurence Rozé, and Yannick Pencolé. Incremental decentralized diagnosis approach for the supervision of a telecommunication network. In *Proc. 12th Intl Workshop on Principles of Diagnosis, DX01*, 2001.

[Hamscher *et al.*, 1992] W. Hamscher, L. Console, and J. de Kleer. *Readings in Model-Based Diagnosis*. Morgan Kaufmann Publishers, 1992.

[Larsson *et al.*, 2000] M. Larsson, I. Klein, D. Lawesson, and U. Nilsson. Fault isolation in object oriented control systems. In *4th IFAC Symposium On Fault Detection, Supervision and Safety for Technical Processes (SAFEPROCESS 2000)*, June 2000.

[Larsson, 1999] M. Larsson. *Behavioral and Structural Model Based Approaches to Discrete Diagnosis*. PhD thesis no 608, Department of Electrical Engineering, Linköping University, 1999.

[Lawesson *et al.*, 2001] Dan Lawesson, Ulf Nilsson, and Inger Klein. Model-checking based fault isolation in uml. In *Proc. 12th Intl Workshop on Principles of Diagnosis, DX01*, pages 103–110, 2001.

[Lawesson, 2000] Dan Lawesson. *Towards Behavioral Model Fault Isolation for Object Oriented Control Systems*. Licentiate thesis no 863, Dept of Computer and Information Science, Linköping University, 2000.

[Lilius and Porres, 1999] J. Lilius and I. Porres. The semantics of UML state machines. Technical Report 293, Turku Centre for Computer Science, 1999.

[Milner, 1989] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Object Management Group, 1999] Object Management Group. OMG Unified Modeling Language Specification, version 1.3, June 1999.

[Reiter, 1987] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, April 1987.

[Sagonas *et al.*, 1994] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, pages 442–453, 1994.

[Sampath *et al.*, 1995] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of Discrete-Event Systems. *IEEE trans. Automatic Control*, 40(9):1555–1575, 1995.

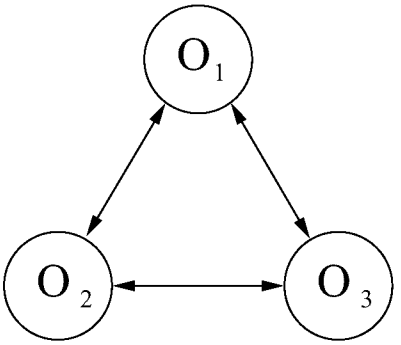

Figure 3: A global picture of the example system consisting of the objects o_1, o_2 and o_3. Each object has behavior as described in Figure 4.

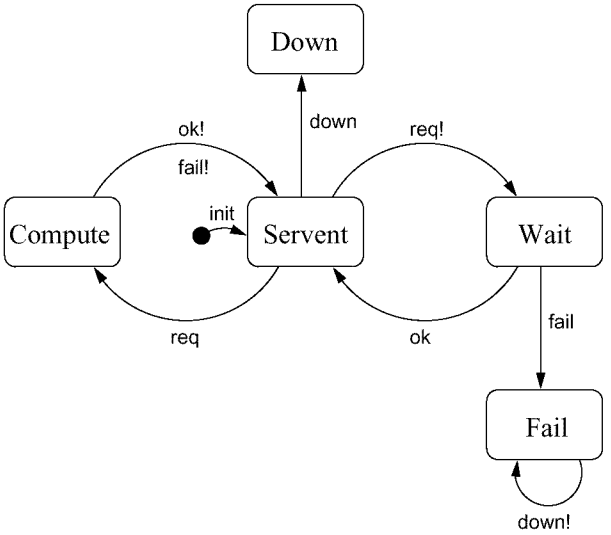

Figure 4: An automata describing a peer-to-peer system. Sending actions are suffixed with ! and the rest of the actions are receiving actions. There are no internal actions in this automata.

$$
\begin{array}{l}
Propagator_n = (), \{ \\
\quad Main \quad \stackrel{def}{=} \quad init(\mathbf{x}).OK(\mathbf{x}) \\
\quad OK(\mathbf{x}) \quad \stackrel{def}{=} \quad fail().Failing(\mathbf{x}) \\
\quad Failing(\mathbf{x}) \quad \stackrel{def}{=} \quad log.Failed(\mathbf{x}) + nolog.Failed(\mathbf{x}) \\
\quad Failed(x_1, x_2, ..., x_n) \quad \stackrel{def}{=} \quad x_1{:}fail().Failed(\mathbf{x}) + ... + x_n{:}fail().Failed(\mathbf{x}) \\
\}
\end{array}
$$

$$
\begin{array}{l}
Breakable = (Propagator, \{\}), \{ \\
\quad OK(\mathbf{x}) \quad \stackrel{def}{=} \quad crit.Failing(\mathbf{x}) \\
\}
\end{array}
$$

Figure 5: Definitions of the classes *Propagator* and *Breakable*